ECE 2300 Digital Logic and Computer Organization Fall 2024

Topic 4: Combinational Building Blocks

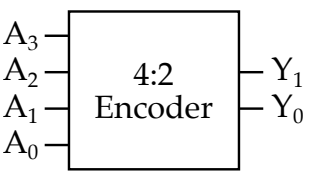
School of Electrical and Computer Engineering
Cornell University

revision: 2024-09-24-09-04

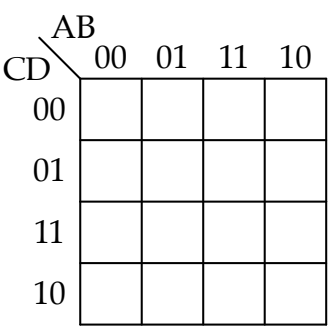
1. Encoders and Decoders

- *Binary encoder* has 2^N inputs and N outputs; sets the output value to reflect which input is one assuming exactly one input is one (encodes a one-value as a binary value)
- *Binary decoder* has N inputs and 2^N outputs and asserts exactly one output depending on the input value (decodes a binary value into a one-hot value)

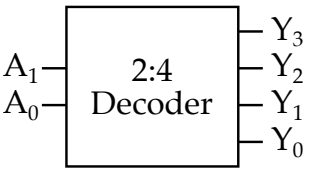
1.1. Encoders



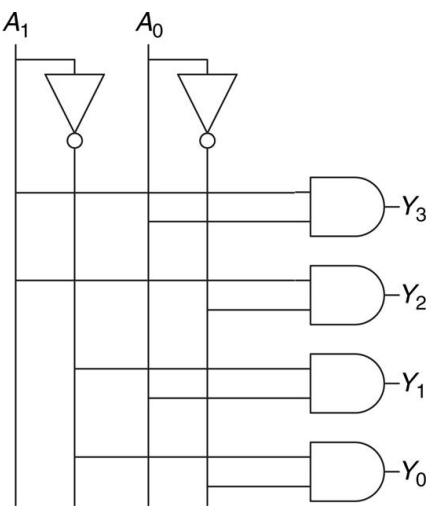
A_3	A_2	A_1	A_0	Y_1	Y_0
0	0	0	1		
0	0	1	0		
0	1	0	0		
1	0	0	0		



1.2. Decoders

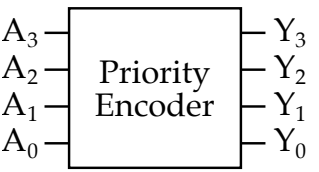


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0				
0	1				
1	0				
1	1				



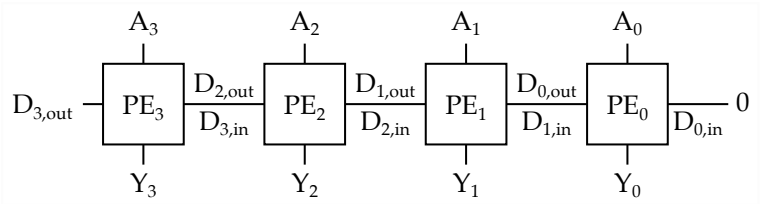
2. Priority Encoders

- *Priority encoder* indicates the first input that is one (i.e., the highest priority input) using a one-hot output

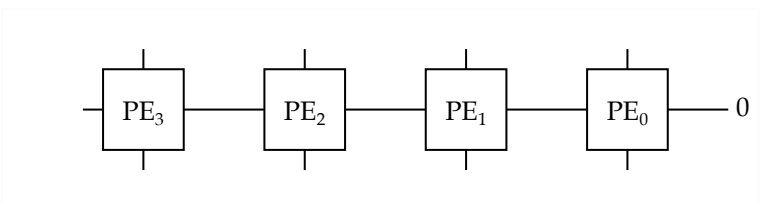


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	0	1
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1
1	0	1	0	0	0	1	0
1	0	1	1	0	0	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

- We can leverage abstraction to manage complexity



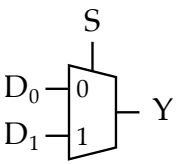
A	D_{in}	Y	D_{out}
0	0		
0	1		
1	0		
1	1		

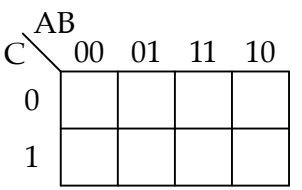


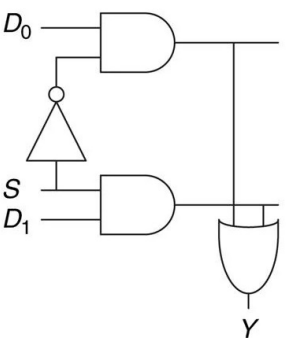
3. Multiplexors and Demultiplexors

- *Multiplexor* chooses among several possible inputs based on the value of a *select* signal
- *Demultiplexor* "routes" an input to one of many several possible outputs based on a *select* signal

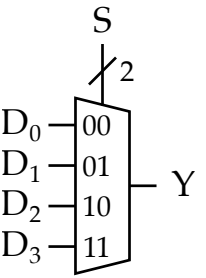
3.1. Multiplexors

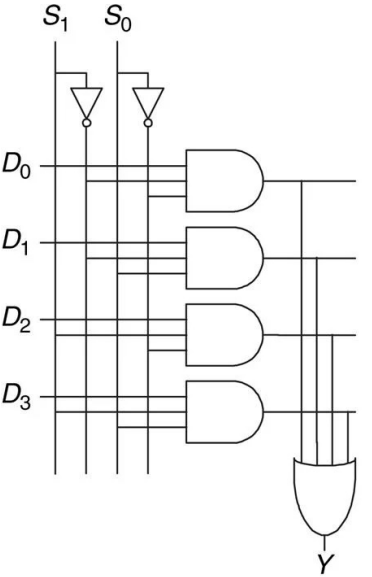


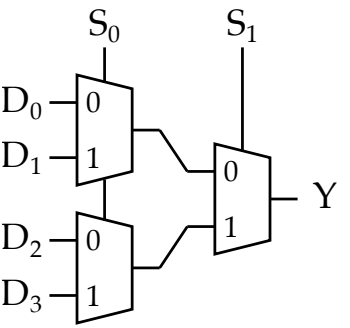




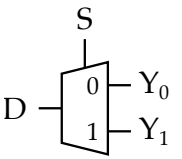
S	D_1	D_0	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	


S
2
D0
D1
D2
D3
00
01
10
11
Y


S1
S0
D0
D1
D2
D3
Y


S0
S1
D0
D1
D2
D3
0
1
0
1
0
1
Y

3.2. Demultiplexors

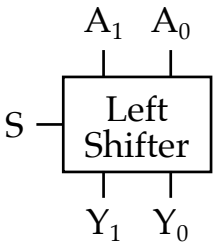


S	D	Y_0	Y_1
0	0		
0	1		
1	0		
1	1		

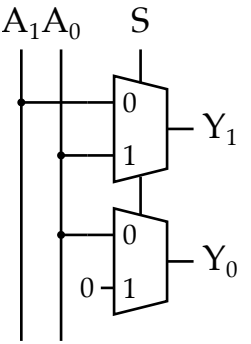
4. Shifters and Rotators

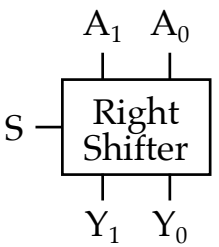
- *Shifter* shifts the input value either left or right by a variable amount specified with a *shift amount* signal; zeros are shifted in
- *Rotator* rotates the input value either left or right by a variable amount specified with a *rotate amount* signal

4.1. Shifters

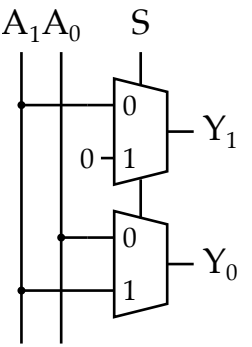


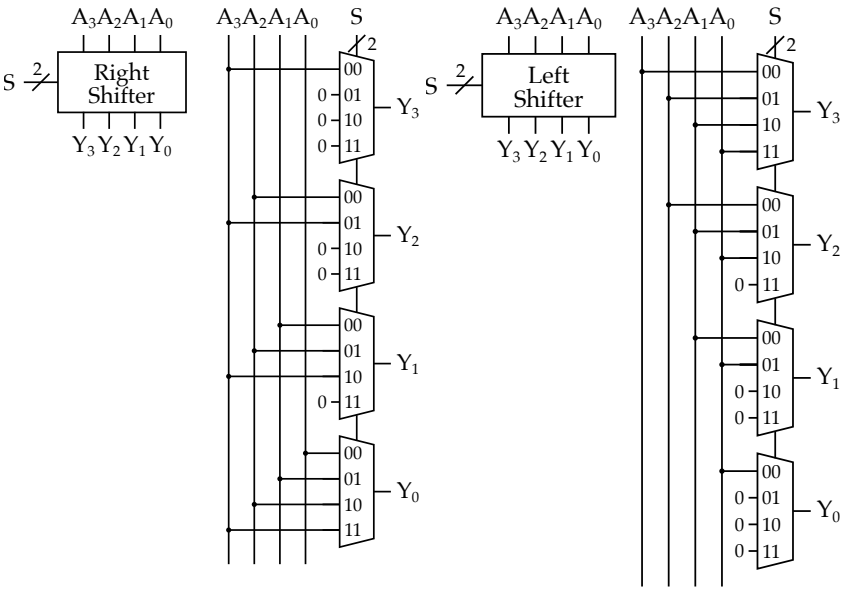
S	A_1	A_0	Y_1	Y_0
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



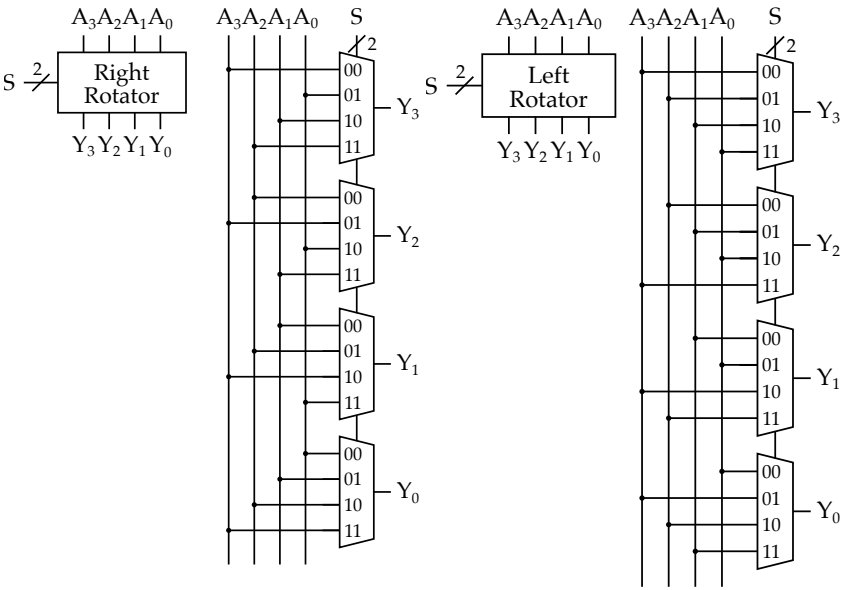


S	A_1	A_0	Y_1	Y_0
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		




A3A2A1A0
S
2
Right Shifter
Y3Y2Y1Y0
A3A2A1A0
S
2
00
01
10
11
0
Y3
Y2
Y1
Y0
A3A2A1A0
S
2
Left Shifter
Y3Y2Y1Y0
A3A2A1A0
S
2
00
01
10
11
0
Y3
Y2
Y1
Y0

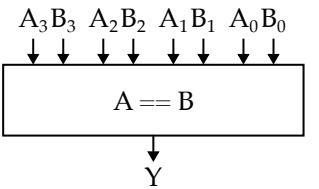
4.2. Rotators


A3A2A1A0
S
2
Right Rotator
Y3 Y2 Y1 Y0
A3A2A1A0
S
2
00
01
10
11
Y3
Y2
Y1
Y0
A3A2A1A0
S
2
Left Rotator
Y3 Y2 Y1 Y0
A3A2A1A0
S
2
00
01
10
11
Y3
Y2
Y1
Y0

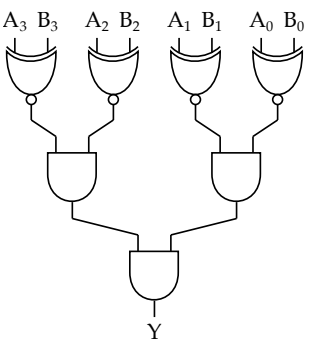
5. Comparators

- *Comparator* compares two binary numbers and outputs whether these numbers are equal, if one is greater than the other, or if one is less than the other.

5.1. Equality Comparator

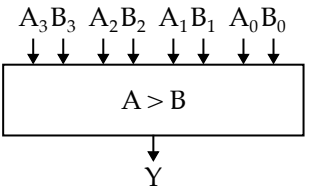


A	B	Y
0	0	
0	1	
1	0	
1	1	

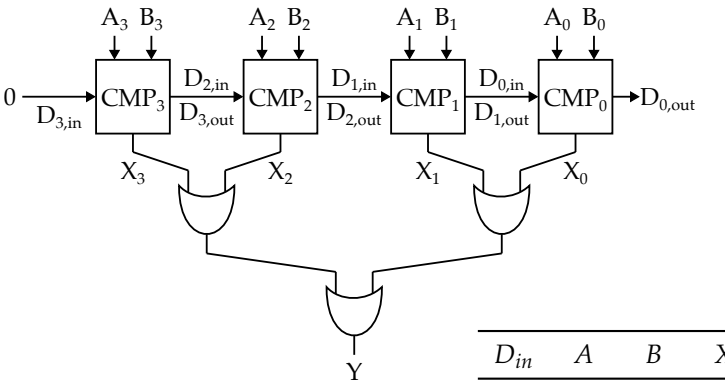


Gate	t_{pd}	t_{cd}
AND2	3τ	3τ
XNOR2	7τ	7τ

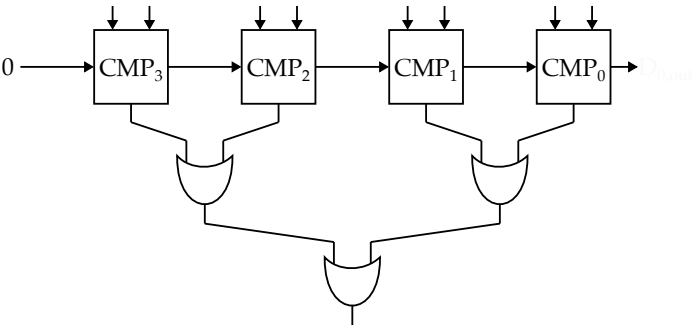
5.2. Greater-Than Comparator



0010 (2)	0101 (5)	0111 (7)
0001 (1)	0110 (6)	0101 (5)
Y =	Y =	Y =
1010 (10)	1110 (14)	1010 (10)
1101 (13)	1011 (11)	1010 (10)
Y =	Y =	Y =



D_{in}	A	B	X	D_{out}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



Gate	t_{pd}	t_{cd}
NOT	1τ	1τ
AND2	3τ	3τ
OR2	4τ	4τ
XOR2	7τ	7τ

Path	Propagation Delay
$A \rightarrow X$	
$B \rightarrow X$	
$D_{in} \rightarrow X$	
$A \rightarrow D_{out}$	
$B \rightarrow D_{out}$	
$D_{in} \rightarrow D_{out}$	

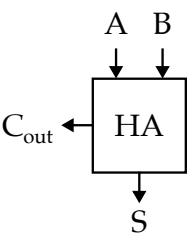
6. Adders and Subtractors

- We will gradually build up complexity to explore a variety of different adder designs

```
  0001      0010      0010      0010      0110
+ 0001    + 0010    + 0010    + 0010    + 0011
------    ------    ------    ------    ------
```

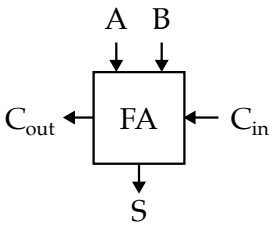
6.1. Half and Full Adders

- *Half adder* adds two one-bit numbers



A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

- *Full adder* adds three one-bit numbers



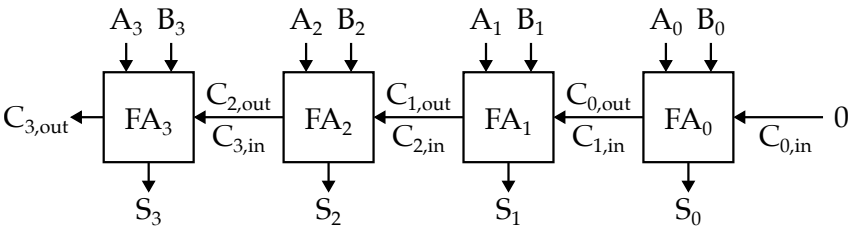
A	B	C_{in}	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

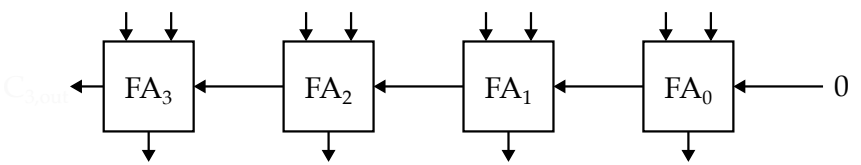
Gate	t_{pd}	t_{cd}
NOT	1τ	1τ
AND2	3τ	3τ
OR2	4τ	4τ
XOR2	7τ	7τ

Path	**Propagation Delay**
$A \rightarrow S$	
$B \rightarrow S$	
$C_{in} \rightarrow S$	
$A \rightarrow C_{out}$	
$B \rightarrow C_{out}$	
$C_{in} \rightarrow C_{out}$	

6.2. Ripple-Carry Adders

- *Ripple-carry adder* chains full adders together

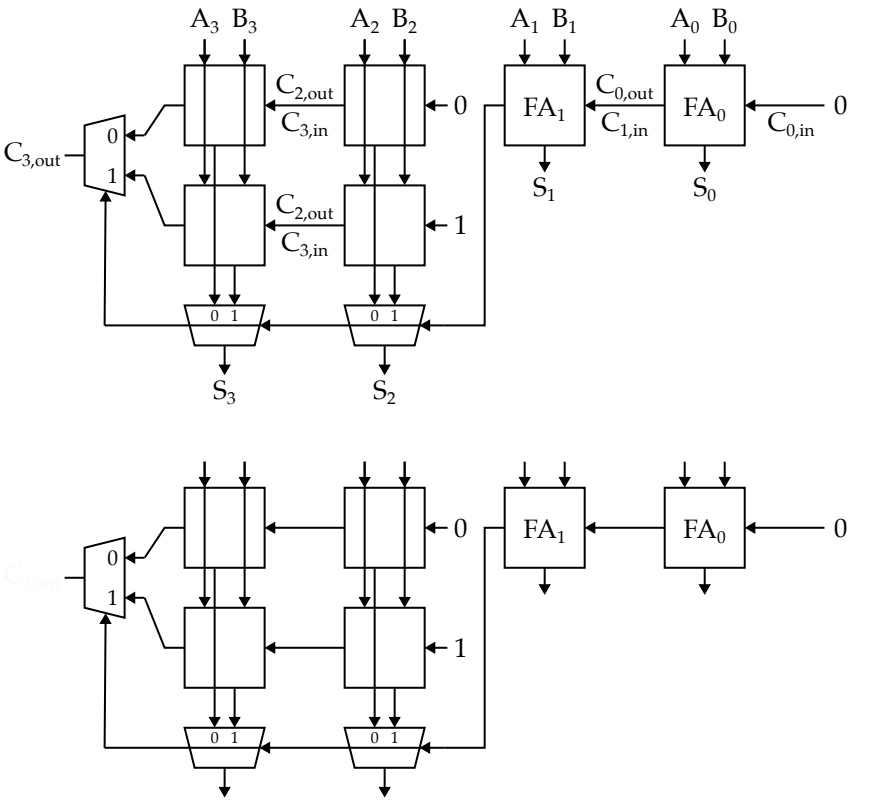


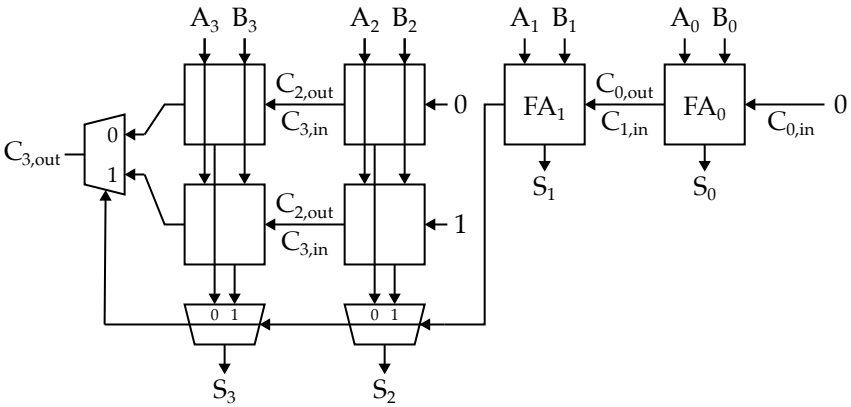


Path	Propagation Delay
$A_0 \rightarrow S_3$	
$B_0 \rightarrow S_3$	
$A_0 \rightarrow C_{3,out}$	
$B_0 \rightarrow C_{3,out}$	

6.3. Carry-Select Adders

- *Carry-select adder* computes most-significant bits twice: once assuming middle of carry chain is zero, once assuming middle of carry chain is one; then selects correct output once middle of charry chain is known





Mux Propagation Path Delays

Path	Propagation Delay
$S \rightarrow Y$	10τ
$D_0 \rightarrow Y$	9τ
$D_1 \rightarrow Y$	9τ

Path	Propagation Delay
$A_0 \rightarrow S_3$	
$B_0 \rightarrow S_3$	
$A_0 \rightarrow C_{3,out}$	
$B_0 \rightarrow C_{3,out}$	

6.4. Carry-Lookahead Adders

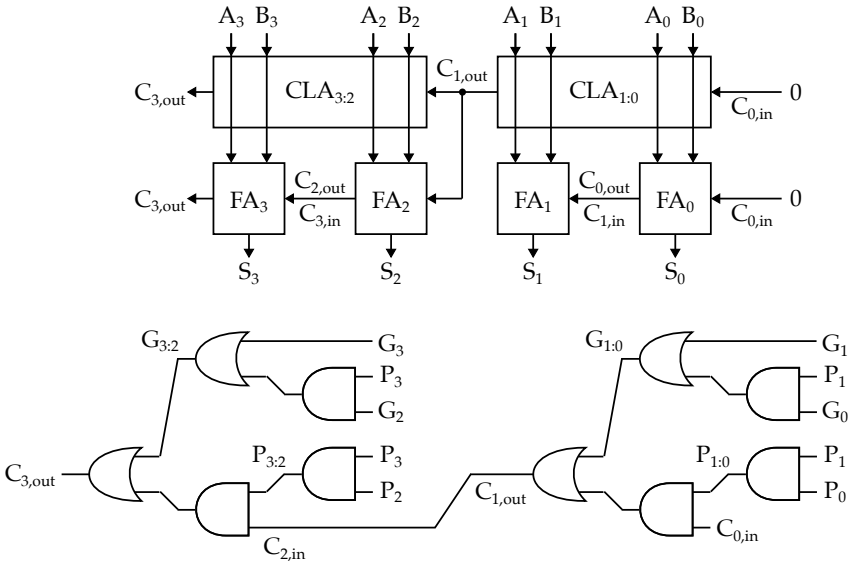
- *Carry-lookahead adder* computes a *generate* and *propagate* signal for blocks of bits in parallel; then quickly computes the carry signal for every block before finally computing the sum in parallel
- Bit-slice *i generates* a carry if it produces a carry output independent of the carry input ($G_i = A_i B_i$)
- Bit-slice *i propagates* a carry if it produces a carry output whenever there is a carry input ($P_i = A_i + B_i$)

$$\begin{aligned} C_i &= A_i B_i + A_i C_i + B_i C_{i-1} \\ &= A_i B_i + (A_i + B_i) C_{i-1} \\ &= G_i + P_i C_{i-1} \end{aligned}$$

- Block $i : j$ *generates* a carry if the block produces a carry output ($C_{i,out}$) independent of the block's carry input ($C_{j,in}$)
- Block $i : j$ *propagates* a carry if the block produces a carry output ($C_{i,out}$) whenever there is a carry input for the block ($C_{j,in}$)
- Example of a block with two bit slices

$$\begin{aligned} C_{i-1} &= G_{i-1} + P_{i-1} C_{i-2} \\ C_i &= G_i + P_i C_{i-1} \\ &= G_i + P_i (G_{i-1} + P_{i-1} C_{i-2}) \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-2} \\ &= (G_i + P_i G_{i-1}) + (P_i P_{i-1}) C_{i-2} \\ &= G_{i:j} + P_{i:j} C_{i-2} \end{aligned}$$

- *Key Idea:* We can compute all of the generate and propagate signals without knowing the carry in; then once we know the carry in we can quickly calculate the carry out



Gate	t_{pd}	t_{cd}
NOT	1τ	1τ
AND2	3τ	3τ
OR2	4τ	4τ
XOR2	7τ	7τ

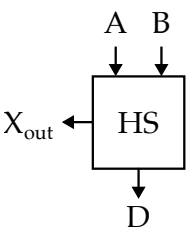
Path	Propagation Delay
$A_0 \rightarrow S_3$	
$B_0 \rightarrow S_3$	
$A_0 \rightarrow C_{3,out}$	
$B_0 \rightarrow C_{3,out}$	

6.5. Subtractors

```
  0001      0011      0010      0100      0100
- 0001    - 0001    - 0001    - 0001    - 0011
------    ------    ------    ------    ------
```

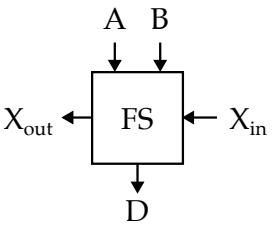
6.6. Half and Full Subtractors

- *Half subtractor* subtracts two one-bit numbers



A	B	X_{out}	D
0	0		
0	1		
1	0		
1	1		

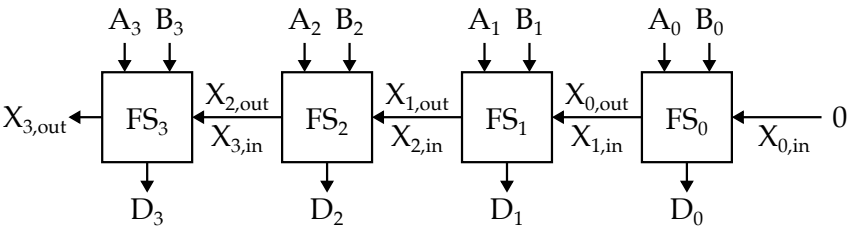
- *Full subtractor* subtracts two one-bit numbers with borrow in

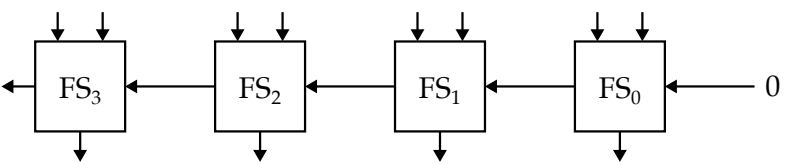


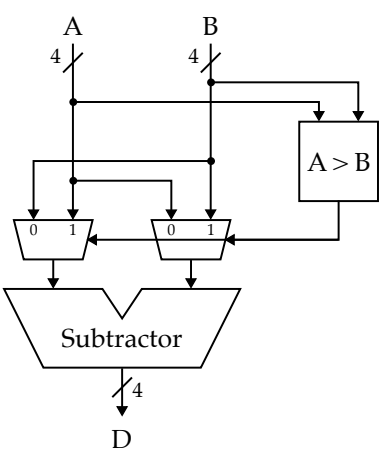
X_{in}	A	B	X_{out}	D
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

6.7. Ripple-Carry Subtractor

- *Ripple-carry subtractor* chains full subtractors together

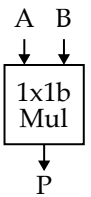






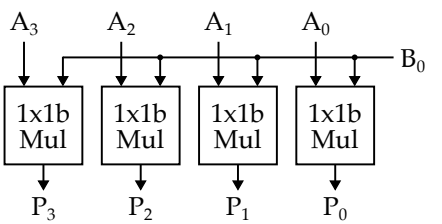
7. Multipliers

7.1. 1x1b Multiplier



A	*B*	*P*
0	0	
0	1	
1	0	
1	1	

7.2. 1x4b Multiplier



7.3. 4x4b Multiplier